**Array methods**

Arrays provide a lot of methods. To make things easier, in this chapter they are split into groups.

**[Add/remove items](https://javascript.info/array-methods" \l "add-remove-items)**

We already know methods that add and remove items from the beginning or the end:

* arr.push(...items) – adds items to the end,
* arr.pop() – extracts an item from the end,
* arr.shift() – extracts an item from the beginning,
* arr.unshift(...items) – adds items to the beginning.

Here are few others.

**[splice](https://javascript.info/array-methods" \l "splice)**

How to delete an element from the array?

The arrays are objects, so we can try to use delete:

let arr = ["I", "go", "home"];

delete arr[1]; // remove "go"

alert( arr[1] ); // undefined

// now arr = ["I", , "home"];

alert( arr.length ); // 3

The element was removed, but the array still has 3 elements, we can see that arr.length == 3.

That’s natural, because delete obj.key removes a value by the key. It’s all it does. Fine for objects. But for arrays we usually want the rest of elements to shift and occupy the freed place. We expect to have a shorter array now.

So, special methods should be used.

The [arr.splice(str)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice) method is a swiss army knife for arrays. It can do everything: insert, remove and replace elements.

The syntax is:

arr.splice(index[, deleteCount, elem1, ..., elemN])

It starts from the position index: removes deleteCount elements and then inserts elem1, ..., elemN at their place. Returns the array of removed elements.

This method is easy to grasp by examples.

Let’s start with the deletion:

let arr = ["I", "study", "JavaScript"];

arr.splice(1, 1); // from index 1 remove 1 element

alert( arr ); // ["I", "JavaScript"]

Easy, right? Starting from the index 1 it removed 1 element.

In the next example we remove 3 elements and replace them with the other two:

let arr = ["I", "study", "JavaScript", "right", "now"];

// remove 3 first elements and replace them with another

arr.splice(0, 3, "Let's", "dance");

alert( arr ) // now ["Let's", "dance", "right", "now"]

Here we can see that splice returns the array of removed elements:

let arr = ["I", "study", "JavaScript", "right", "now"];

// remove 2 first elements

let removed = arr.splice(0, 2);

alert( removed ); // "I", "study" <-- array of removed elements

The splice method is also able to insert the elements without any removals. For that we need to set deleteCount to 0:

let arr = ["I", "study", "JavaScript"];

// from index 2

// delete 0

// then insert "complex" and "language"

arr.splice(2, 0, "complex", "language");

alert( arr ); // "I", "study", "complex", "language", "JavaScript"

**Negative indexes allowed**

Here and in other array methods, negative indexes are allowed. They specify the position from the end of the array, like here:

let arr = [1, 2, 5];

// from index -1 (one step from the end)

// delete 0 elements,

// then insert 3 and 4

arr.splice(-1, 0, 3, 4);

alert( arr ); // 1,2,3,4,5

**[slice](https://javascript.info/array-methods" \l "slice)**

The method [arr.slice](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/slice) is much simpler than similar-looking arr.splice.

The syntax is:

arr.slice(start, end)

It returns a new array containing all items from index "start" to "end" (not including "end"). Both startand end can be negative, in that case position from array end is assumed.

It works like str.slice, but makes subarrays instead of substrings.

For instance:

let str = "test";

let arr = ["t", "e", "s", "t"];

alert( str.slice(1, 3) ); // es

alert( arr.slice(1, 3) ); // e,s

alert( str.slice(-2) ); // st

alert( arr.slice(-2) ); // s,t

**[concat](https://javascript.info/array-methods" \l "concat)**

The method [arr.concat](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/concat) joins the array with other arrays and/or items.

The syntax is:

arr.concat(arg1, arg2...)

It accepts any number of arguments – either arrays or values.

The result is a new array containing items from arr, then arg1, arg2 etc.

If an argument is an array or has Symbol.isConcatSpreadable property, then all its elements are copied. Otherwise, the argument itself is copied.

For instance:

let arr = [1, 2];

// merge arr with [3,4]

alert( arr.concat([3, 4])); // 1,2,3,4

// merge arr with [3,4] and [5,6]

alert( arr.concat([3, 4], [5, 6])); // 1,2,3,4,5,6

// merge arr with [3,4], then add values 5 and 6

alert( arr.concat([3, 4], 5, 6)); // 1,2,3,4,5,6

Normally, it only copies elements from arrays (“spreads” them). Other objects, even if they look like arrays, added as a whole:

let arr = [1, 2];

let arrayLike = {

0: "something",

length: 1

};

alert( arr.concat(arrayLike) ); // 1,2,[object Object]

//[1, 2, arrayLike]

…But if an array-like object has Symbol.isConcatSpreadable property, then its elements are added instead:

let arr = [1, 2];

let arrayLike = {

0: "something",

1: "else",

[Symbol.isConcatSpreadable]: true,

length: 2

};

alert( arr.concat(arrayLike) ); // 1,2,something,else

**[Iterate: forEach](https://javascript.info/array-methods" \l "iterate-foreach)**

The [arr.forEach](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach) method allows to run a function for every element of the array.

The syntax:

arr.forEach(function(item, index, array) {

// ... do something with item

});

For instance, this shows each element of the array:

// for each element call alert

["Bilbo", "Gandalf", "Nazgul"].forEach(alert);

And this code is more elaborate about their positions in the target array:

["Bilbo", "Gandalf", "Nazgul"].forEach((item, index, array) => {

alert(`${item} is at index ${index} in ${array}`);

});

The result of the function (if it returns any) is thrown away and ignored.

**[Searching in array](https://javascript.info/array-methods" \l "searching-in-array)**

These are methods to search for something in an array.

**[indexOf/lastIndexOf and includes](https://javascript.info/array-methods" \l "indexof-lastindexof-and-includes)**

The methods [arr.indexOf](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/indexOf), [arr.lastIndexOf](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/lastIndexOf) and [arr.includes](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/includes) have the same syntax and do essentially the same as their string counterparts, but operate on items instead of characters:

* arr.indexOf(item, from) looks for item starting from index from, and returns the index where it was found, otherwise -1.
* arr.lastIndexOf(item, from) – same, but looks for from right to left.
* arr.includes(item, from) – looks for item starting from index from, returns true if found.

For instance:

let arr = [1, 0, false];

alert( arr.indexOf(0) ); // 1

alert( arr.indexOf(false) ); // 2

alert( arr.indexOf(null) ); // -1

alert( arr.includes(1) ); // true

Note that the methods use === comparison. So, if we look for false, it finds exactly false and not the zero.

If we want to check for inclusion, and don’t want to know the exact index, then arr.includes is preferred.

Also, a very minor difference of includes is that it correctly handles NaN, unlike indexOf/lastIndexOf:

const arr = [NaN];

alert( arr.indexOf(NaN) ); // -1 (should be 0, but === equality doesn't work for NaN)

alert( arr.includes(NaN) );// true (correct)

**[find and findIndex](https://javascript.info/array-methods" \l "find-and-findindex)**

Imagine we have an array of objects. How do we find an object with the specific condition?

Here the [arr.find](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/find) method comes in handy.

The syntax is:

let result = arr.find(function(item, index, array) {

// if true is returned, item is returned and iteration is stopped

// for falsy scenario returns undefined

});

The function is called repetitively for each element of the array:

* item is the element.
* index is its index.
* array is the array itself.

If it returns true, the search is stopped, the item is returned. If nothing found, undefined is returned.

For example, we have an array of users, each with the fields id and name. Let’s find the one with id == 1:

let users = [

{id: 1, name: "John"},

{id: 2, name: "Pete"},

{id: 3, name: "Mary"}

];

let user = users.find(item => item.id == 1);

alert(user.name); // John

In real life arrays of objects is a common thing, so the find method is very useful.

Note that in the example we provide to find the function item => item.id == 1 with one argument. Other arguments of this function are rarely used.

The [arr.findIndex](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/findIndex) method is essentially the same, but it returns the index where the element was found instead of the element itself and -1 is returned when nothing is found.

**[filter](https://javascript.info/array-methods" \l "filter)**

The find method looks for a single (first) element that makes the function return true.

If there may be many, we can use [arr.filter(fn)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter).

The syntax is similar to find, but filter continues to iterate for all array elements even if true is already returned:

let results = arr.filter(function(item, index, array) {

// if true item is pushed to results and iteration continues

// returns empty array for complete falsy scenario

});

For instance:

let users = [

{id: 1, name: "John"},

{id: 2, name: "Pete"},

{id: 3, name: "Mary"}

];

// returns array of the first two users

let someUsers = users.filter(item => item.id < 3);

alert(someUsers.length); // 2

**[Transform an array](https://javascript.info/array-methods" \l "transform-an-array)**

This section is about the methods transforming or reordering the array.

**[map](https://javascript.info/array-methods" \l "map)**

The [arr.map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map) method is one of the most useful and often used.

The syntax is:

let result = arr.map(function(item, index, array) {

// returns the new value instead of item

})

It calls the function for each element of the array and returns the array of results.

For instance, here we transform each element into its length:

let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);

alert(lengths); // 5,7,6

**[sort(fn)](https://javascript.info/array-methods" \l "sort-fn)**

The method [arr.sort](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort) sorts the array *in place*.

For instance:

let arr = [ 1, 2, 15 ];

// the method reorders the content of arr (and returns it)

arr.sort();

alert( arr ); // 1, 15, 2

Did you notice anything strange in the outcome?

The order became 1, 15, 2. Incorrect. But why?

**The items are sorted as strings by default.**

Literally, all elements are converted to strings and then compared. So, the lexicographic ordering is applied and indeed "2" > "15".

To use our own sorting order, we need to supply a function of two arguments as the argument of arr.sort().

The function should work like this:

function compare(a, b) {

if (a > b) return 1;

if (a == b) return 0;

if (a < b) return -1;

}

For instance:

function compareNumeric(a, b) {

if (a > b) return 1;

if (a == b) return 0;

if (a < b) return -1;

}

let arr = [ 1, 2, 15 ];

arr.sort(compareNumeric);

alert(arr); // 1, 2, 15

Now it works as intended.

Let’s step aside and think what’s happening. The arr can be array of anything, right? It may contain numbers or strings or html elements or whatever. We have a set of *something*. To sort it, we need an *ordering function* that knows how to compare its elements. The default is a string order.

The arr.sort(fn) method has a built-in implementation of sorting algorithm. We don’t need to care how it exactly works (an optimized [quicksort](https://en.wikipedia.org/wiki/Quicksort) most of the time). It will walk the array, compare its elements using the provided function and reorder them, all we need is to provide the fn which does the comparison.

By the way, if we ever want to know which elements are compared – nothing prevents from alerting them:

[1, -2, 15, 2, 0, 8].sort(function(a, b) {

alert( a + " <> " + b );

});

The algorithm may compare an element multiple times in the process, but it tries to make as few comparisons as possible.

**A comparison function may return any number**

Actually, a comparison function is only required to return a positive number to say “greater” and a negative number to say “less”.

That allows to write shorter functions:

let arr = [ 1, 2, 15 ];

arr.sort(function(a, b) { return a - b; });

alert(arr); // 1, 2, 15

**Arrow functions for the best**

Remember [arrow functions](https://javascript.info/function-expressions-arrows#arrow-functions)? We can use them here for neater sorting:

arr.sort( (a, b) => a - b );

This works exactly the same as the other, longer, version above.

**[reverse](https://javascript.info/array-methods" \l "reverse)**

The method [arr.reverse](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reverse) reverses the order of elements in arr.

For instance:

let arr = [1, 2, 3, 4, 5];

arr.reverse();

alert( arr ); // 5,4,3,2,1

It also returns the array arr after the reversal.

**[split and join](https://javascript.info/array-methods" \l "split-and-join)**

Here’s the situation from the real life. We are writing a messaging app, and the person enters the comma-delimited list of receivers: John, Pete, Mary. But for us an array of names would be much more comfortable than a single string. How to get it?

The [str.split(delim)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/split) method does exactly that. It splits the string into an array by the given delimiter delim.

In the example below, we split by a comma followed by space:

let names = 'Bilbo, Gandalf, Nazgul';

let arr = names.split(', ');

for (let name of arr) {

alert( `A message to ${name}.` ); // A message to Bilbo (and other names)

}

The split method has an optional second numeric argument – a limit on the array length. If it is provided, then the extra elements are ignored. In practice it is rarely used though:

let arr = 'Bilbo, Gandalf, Nazgul, Saruman'.split(', ', 2);

alert(arr); // Bilbo, Gandalf

**Split into letters**

The call to split(s) with an empty s would split the string into an array of letters:

let str = "test";

alert( str.split('') ); // t,e,s,t

The call [arr.join(separator)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/join) does the reverse to split. It creates a string of arr items glued by separatorbetween them.

For instance:

let arr = ['Bilbo', 'Gandalf', 'Nazgul'];

let str = arr.join(';');

alert( str ); // Bilbo;Gandalf;Nazgul

**[reduce/reduceRight](https://javascript.info/array-methods" \l "reduce-reduceright)**

When we need to iterate over an array – we can use forEach, for or for..of.

When we need to iterate and return the data for each element – we can use map.

The methods [arr.reduce](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce) and [arr.reduceRight](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduceRight) also belong to that breed, but are a little bit more intricate. They are used to calculate a single value based on the array.

The syntax is:

let value = arr.reduce(function(previousValue, item, index, array) {

// ...

}, initial);

The function is applied to the elements. You may notice the familiar arguments, starting from the 2nd:

* item – is the current array item.
* index – is its position.
* array – is the array.

So far, like forEach/map. But there’s one more argument:

* previousValue – is the result of the previous function call, initial for the first call.

The easiest way to grasp that is by example.

Here we get a sum of array in one line:

let arr = [1, 2, 3, 4, 5];

let result = arr.reduce((sum, current) => sum + current, 0);

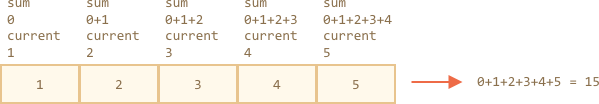
alert(result); // 15

Here we used the most common variant of reduce which uses only 2 arguments.

Let’s see the details of what’s going on.

1. On the first run, sum is the initial value (the last argument of reduce), equals 0, and current is the first array element, equals 1. So the result is 1.
2. On the second run, sum = 1, we add the second array element (2) to it and return.
3. On the 3rd run, sum = 3 and we add one more element to it, and so on…

The calculation flow:



Or in the form of a table, where each row represents a function call on the next array element:

|  | **sum** | **current** | **result** |
| --- | --- | --- | --- |
| the first call | 0 | 1 | 1 |
| the second call | 1 | 2 | 3 |
| the third call | 3 | 3 | 6 |
| the fourth call | 6 | 4 | 10 |
| the fifth call | 10 | 5 | 15 |

As we can see, the result of the previous call becomes the first argument of the next one.

We also can omit the initial value:

let arr = [1, 2, 3, 4, 5];

// removed initial value from reduce (no 0)

let result = arr.reduce((sum, current) => sum + current);

alert( result ); // 15

The result is the same. That’s because if there’s no initial, then reduce takes the first element of the array as the initial value and starts the iteration from the 2nd element.

The calculation table is the same as above, minus the first row.

But such use requires an extreme care. If the array is empty, then reduce call without initial value gives an error.

Here’s an example:

let arr = [];

// Error: Reduce of empty array with no initial value

// if the initial value existed, reduce would return it for the empty arr.

arr.reduce((sum, current) => sum + current);

So it’s advised to always specify the initial value.

The method [arr.reduceRight](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduceRight) does the same, but goes from right to left.

**[Array.isArray](https://javascript.info/array-methods" \l "array-isarray)**

Arrays do not form a separate language type. They are based on objects.

So typeof does not help to distinguish a plain object from an array:

alert(typeof {}); // object

alert(typeof []); // same

…But arrays are used so often that there’s a special method for that: [Array.isArray(value)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/isArray). It returns true if the value is an array, and false otherwise.

alert(Array.isArray({})); // false

alert(Array.isArray([])); // true

**[Most methods support “thisArg”](https://javascript.info/array-methods" \l "most-methods-support-thisarg)**

Almost all array methods that call functions – like find, filter, map, with a notable exception of sort, accept an optional additional parameter thisArg.

That parameter is not explained in the sections above, because it’s rarely used. But for completeness we have to cover it.

Here’s the full syntax of these methods:

arr.find(func, thisArg);

arr.filter(func, thisArg);

arr.map(func, thisArg);

// ...

// thisArg is the optional last argument

The value of thisArg parameter becomes this for func.

For instance, here we use an object method as a filter and thisArg comes in handy:

let user = {

age: 18,

younger(otherUser) {

return otherUser.age < this.age;

}

};

let users = [

{age: 12},

{age: 16},

{age: 32}

];

// find all users younger than user

let youngerUsers = users.filter(user.younger, user);

alert(youngerUsers.length); // 2

In the call above, we use user.younger as a filter and also provide user as the context for it. If we didn’t provide the context, users.filter(user.younger) would call user.younger as a standalone function, with this=undefined. That would mean an instant error.

**[Summary](https://javascript.info/array-methods" \l "summary)**

A cheatsheet of array methods:

* To add/remove elements:
  + push(...items) – adds items to the end,
  + pop() – extracts an item from the end,
  + shift() – extracts an item from the beginning,
  + unshift(...items) – adds items to the beginning.
  + splice(pos, deleteCount, ...items) – at index pos delete deleteCount elements and insert items.
  + slice(start, end) – creates a new array, copies elements from position start till end (not inclusive) into it.
  + concat(...items) – returns a new array: copies all members of the current one and adds items to it. If any of items is an array, then its elements are taken.
* To search among elements:
  + indexOf/lastIndexOf(item, pos) – look for item starting from position pos, return the index or -1if not found.
  + includes(value) – returns true if the array has value, otherwise false.
  + find/filter(func) – filter elements through the function, return first/all values that make it return true.
  + findIndex is like find, but returns the index instead of a value.
* To iterate over elements:
  + forEach(func) – calls func for every element, does not return anything.
* To transform the array:
  + map(func) – creates a new array from results of calling func for every element.
  + sort(func) – sorts the array in-place, then returns it.
  + reverse() – reverses the array in-place, then returns it.
  + split/join – convert a string to array and back.
  + reduce(func, initial) – calculate a single value over the array by calling func for each element and passing an intermediate result between the calls.
* Additionally:
  + Array.isArray(arr) checks arr for being an array.

Please note that methods sort, reverse and splice modify the array itself.

These methods are the most used ones, they cover 99% of use cases. But there are few others:

* [arr.some(fn)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/some)/[arr.every(fn)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/every) checks the array.

The function fn is called on each element of the array similar to map. If any/all results are true, returns true, otherwise false.

* [arr.fill(value, start, end)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/fill) – fills the array with repeating value from index start to end.
* [arr.copyWithin(target, start, end)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/copyWithin) – copies its elements from position start till position end into *itself*, at position target (overwrites existing).

For the full list, see the [manual](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array).

From the first sight it may seem that there are so many methods, quite difficult to remember. But actually that’s much easier than it seems.

Look through the cheatsheet just to be aware of them. Then solve the tasks of this chapter to practice, so that you have experience with array methods.

Afterwards whenever you need to do something with an array, and you don’t know how – come here, look at the cheatsheet and find the right method. Examples will help you to write it correctly. Soon you’ll automatically remember the methods, without specific efforts from your side.